Automatically Selecting the Number of Aggregators for Collective I/O Operations

Mohamad Chaarawi and Edgar Gabriel

Parallel Software Technologies Laboratory
Department of Computer Science, University of Houston
<mschaara,gabriel>@cs.uh.edu

Outline

- Motivation
- Automatically determining the number of aggregators
- Experimental Results
- Conclusions and future work

Motivation

- I/O one of the most severe challenges for high-end computing
- MPI 2 introduced the notion of parallel I/O
  - Relaxed consistency semantics
  - Collective I/O
  - Nonblocking I/O
  - File view

Collective I/O operations

- Allows to rearrange data across multiple processes

- Popular algorithm: two-phase I/O
- Algorithm for a collective write operation
  - Step 1:
    - gather data from multiple processes on aggregators
    - Sort data based on the offset in the file
  - Step 2: aggregators write data

Collective I/O operations (II)

- Only a subset of processes actually touch a file (aggregators)
- Large read/write operations split into multiple cycles internally
  - Limits the size of temporary buffers
  - Overlaps communication and I/O operations

- Dynamic segmentation algorithm:
  - Variant of two-phase I/O algorithms
  - Subdivides processes internally into groups
  - One aggregator per group

Two-phase I/O vs. dynamic segmentation

File layout

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Process 0   Process 1   Process 2   Process 3

Two-phase I/O with 2 aggregators

Process 0

1	2	3	4	5	6	7	8

Process 2

9	10	11	12	13	14	15	16

Dynamic segmentation algorithm with 2 aggregators

Process 0

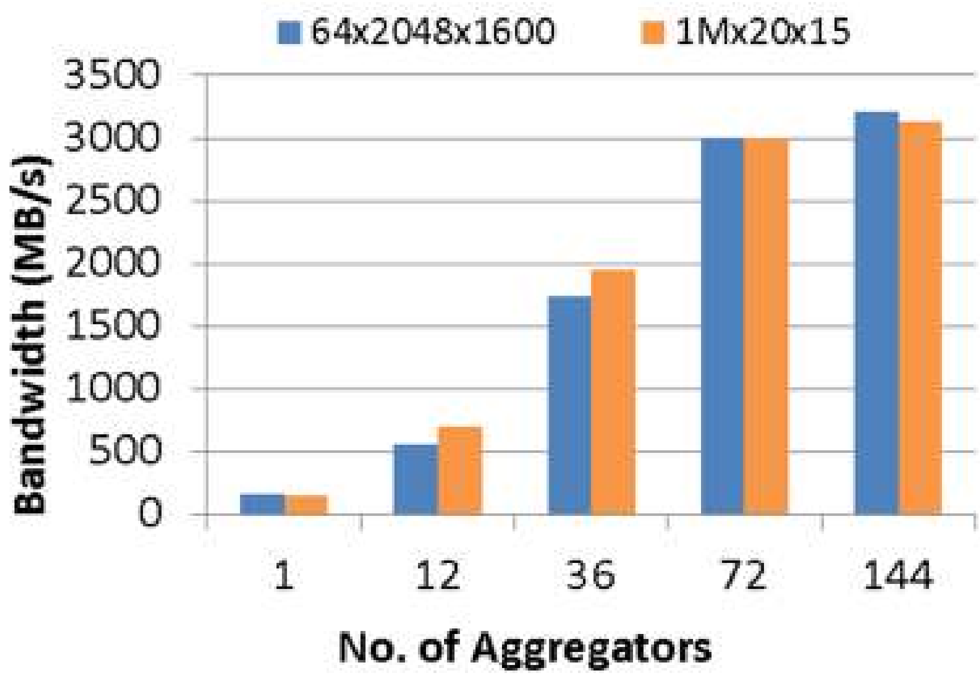
1	2	3	4
9	10	11	12

Process 2

5	6	7	8
13	14	15	16

Performance Considerations

- Performance of Tile I/O benchmark using two-phase I/O using 144 processes on a Lustre file system depending on the number of aggregators



Performance considerations (II)

- Contradicting goals:
  - Generate large consecutive chunks -> fewer aggregators
  - Increase throughput -> more aggregators

- Setting number of aggregators
  - Fixed number: 1, number of processes, number of nodes, number of I/O servers
  - Tune for a particular platform and application

Determining the number of aggregators

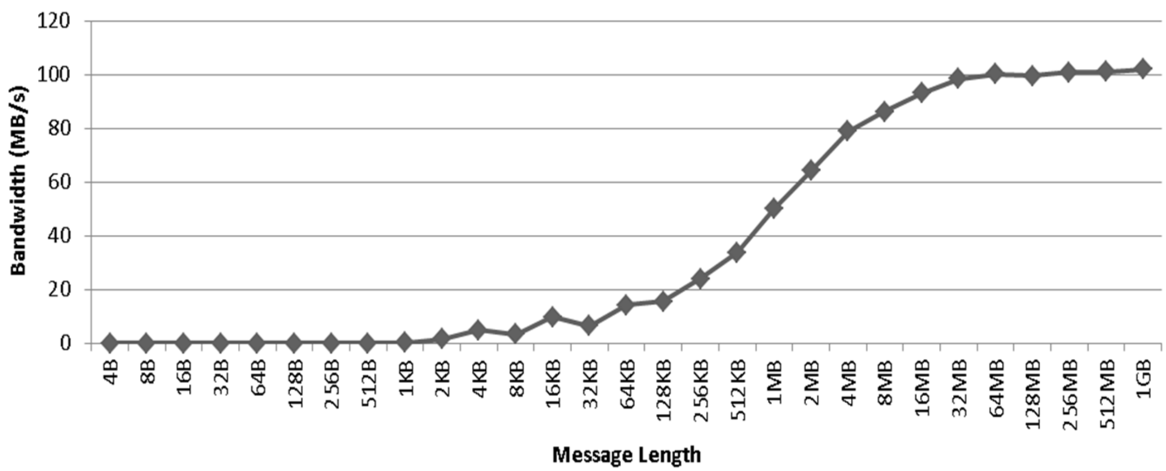
1) Determine the minimum data size *k* for an individual process which leads to maximum write bandwidth

2) Determine initial number of aggregators taking file view and/or process topology into account.

3) Refine the number of aggregators based on the overall amount of data written in the collective call

1. Determining the saturation point

- Loop of individual write operations with increasing data size
  - Avoid caching effects
  - `MPI_File_write()` vs. POSIX `write()`
  - Performed once, e.g. by system administrator
- Saturation point: first element which achieves (close to) maximum bandwidth



2. Initial assignment of aggregators

- Based on fileview
  - Only 2-D pattern handled at this time
  - 1 aggregator per row of processes

Group 1	0	1	2	3
Group 2	4	5	6	7
Group 3	8	9	10	11
Group 4	12	13	14	15

- Based on Cartesian process topology
  - Assumption: process topology related to file access
- Based on hints
  - Not implemented at this time

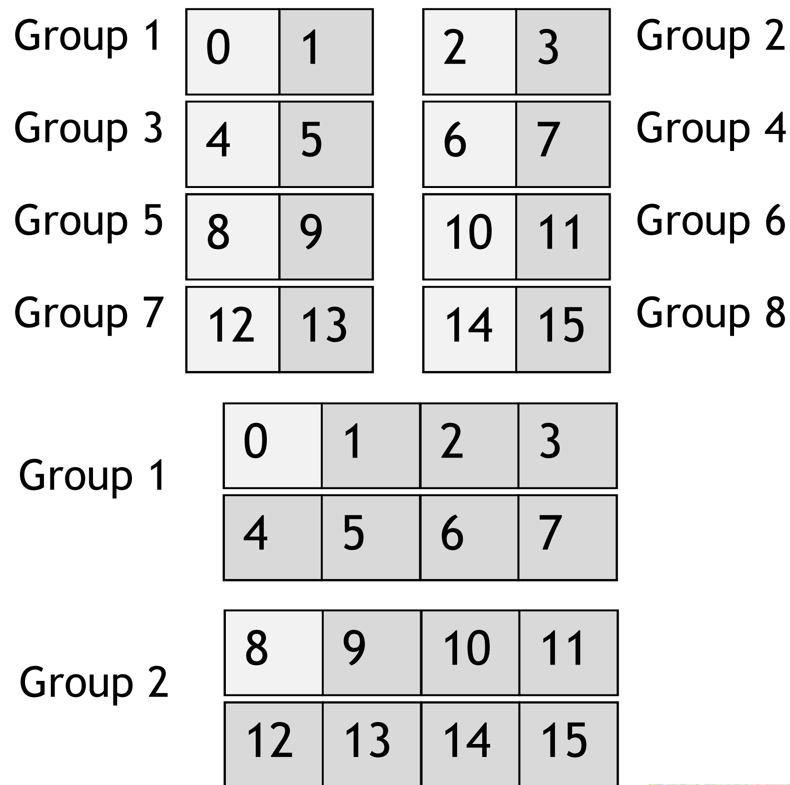
- Without fileview or Cartesian topology:
  - Every process is an aggregator

3. Refinement step

- Based on actual amount of data written across all processes in one collective call

- k < no. of bytes written in group
  -> split group

- k > no. of bytes written in group
  -> merge groups



Discussion of algorithm

- Number of aggregators depends on overall data volume being written
  - Different calls to `MPI_File_write_all` with different data volumes will result in different number of aggregators used

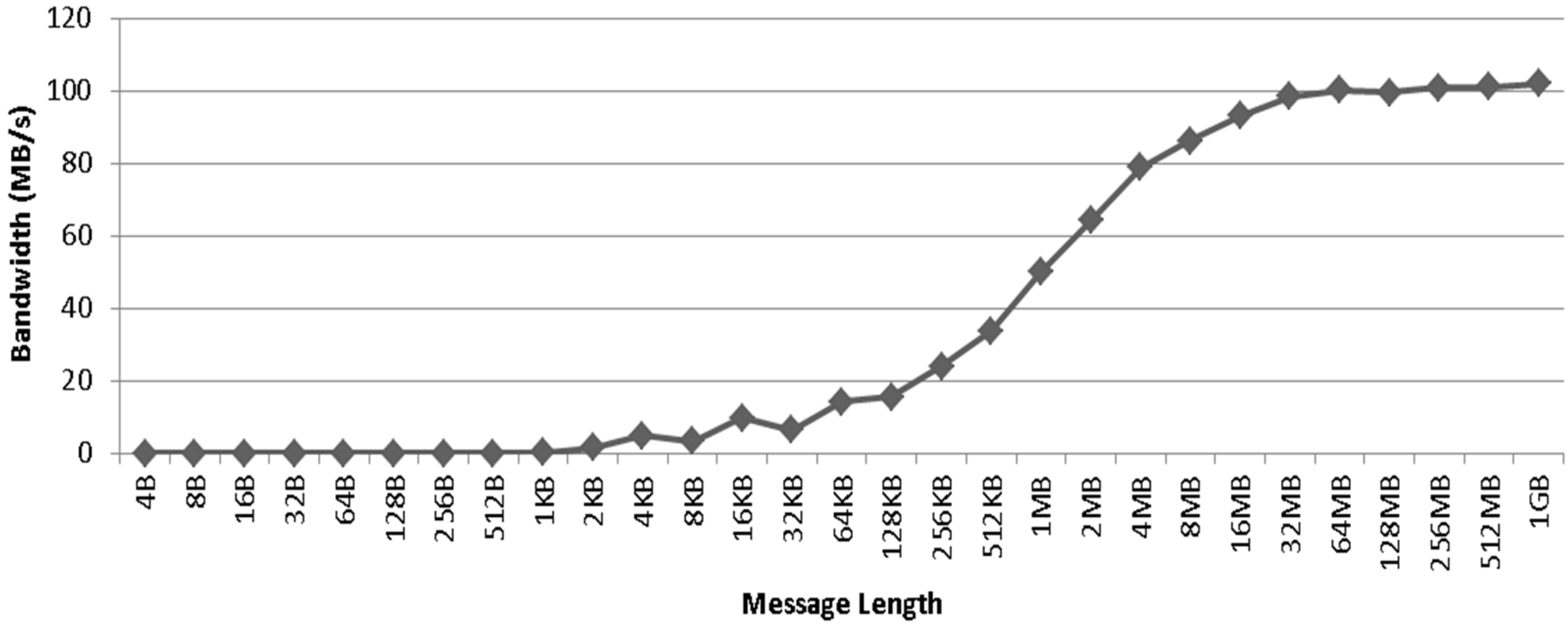
- For fixed problem size, number of aggregators is independent of the number of processes used

- Same approach used for two-phase I/O, dynamic segmentation, and static segmentation

Some performance results

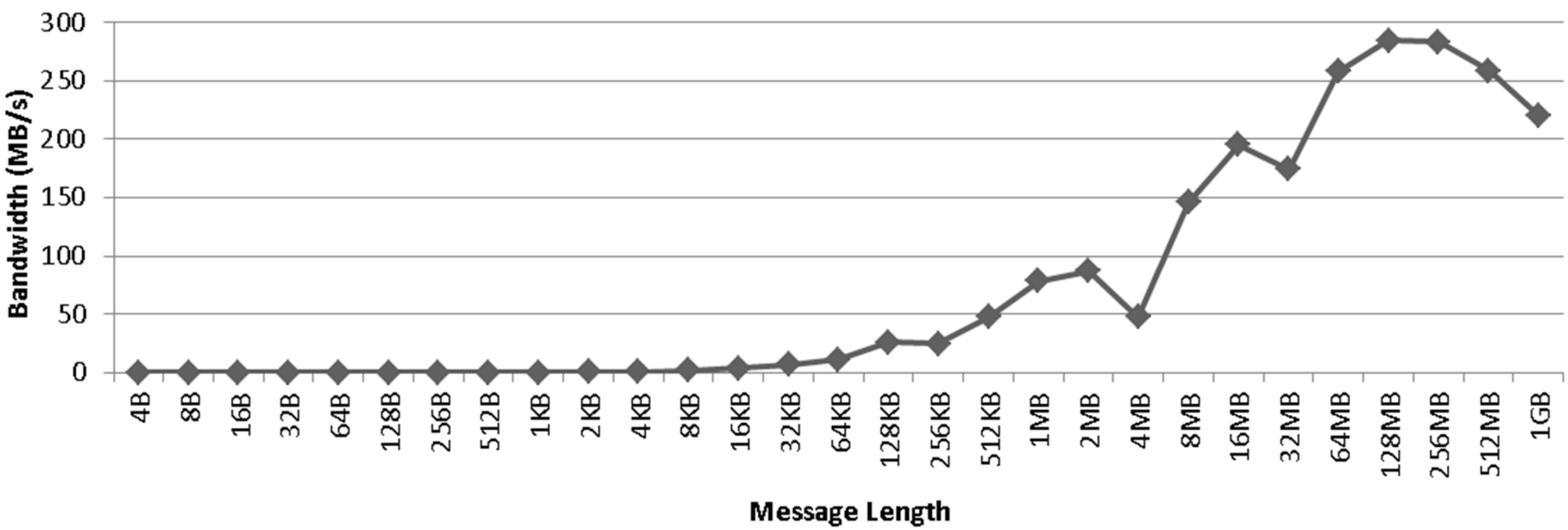
- Shark cluster at University of Houston
  - PVFS2 version 2.8.2
  - 22 disks on 22 nodes, 64 KB stripe size
  - Gigabit Ethernet network used for I/O
  - 29 compute nodes (88 cores)
- Deimos cluster at TU Dresden
  - Lustre file system 1.6.7
  - 11 I/O servers, 48 OSTs, 1 MB stripe size
  - 4X SDR InfiniBand network used for I/O
  - 724 compute nodes (> 2,500 cores)
- Implemented in OMPIO (Open MPI trunk rev. 24428)

Shark saturation point



Saturation point k = 32MB

Deimos saturation point



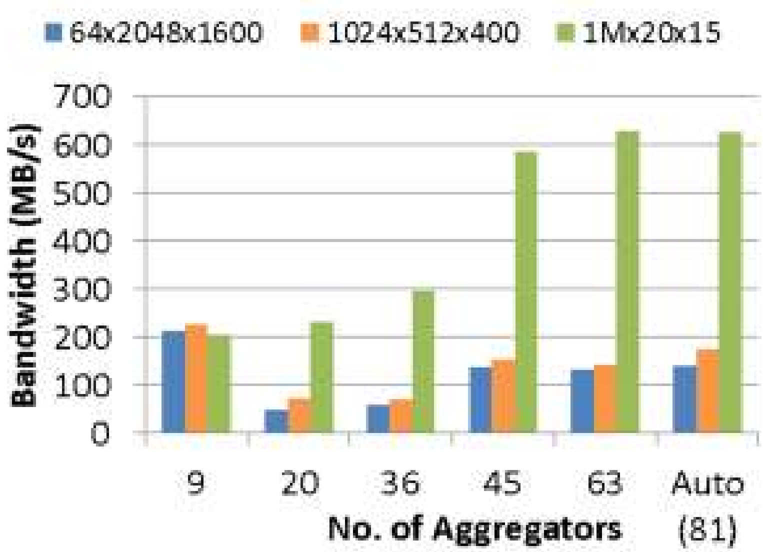
Saturation point k = 128MB

Benchmarks and test cases used

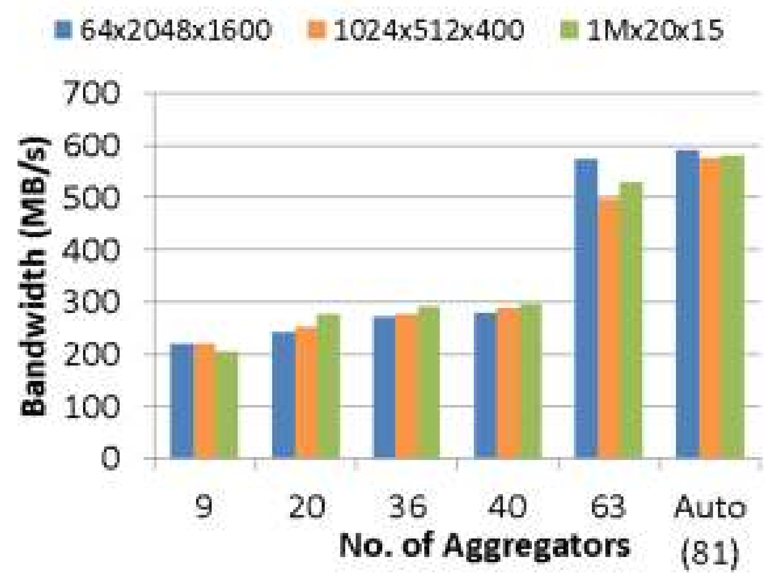
- Tile I/O
  - 2-D access pattern, cartesian communicator
- BT I/O
  - Application benchmark using 2-D access pattern
- Latency I/O
  - Round-robin data distribution across processes
- Image processing application
  - 1-D data distribution

Shark Tile I/O

- 81 processes test case



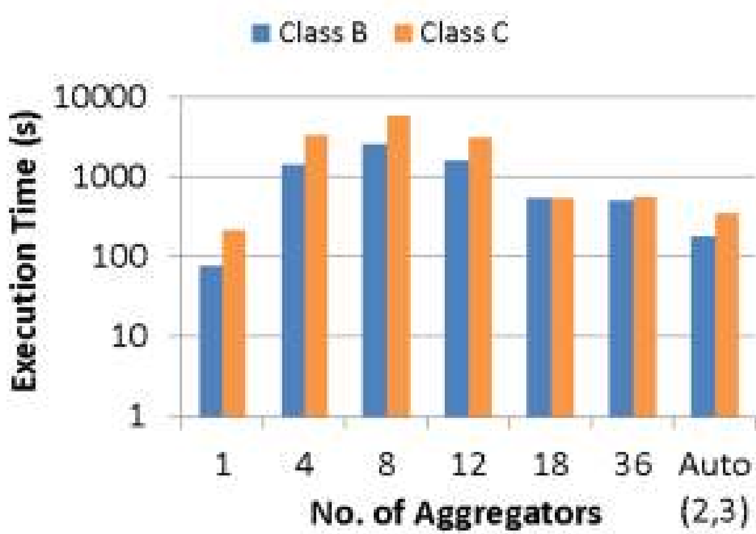
dynamic segmentation



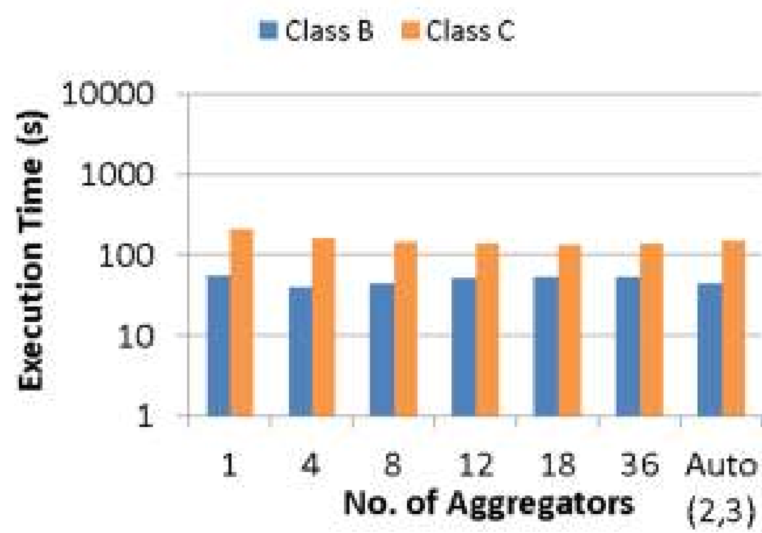
two-phase I/O

Shark BT I/O

- 36 processes test case



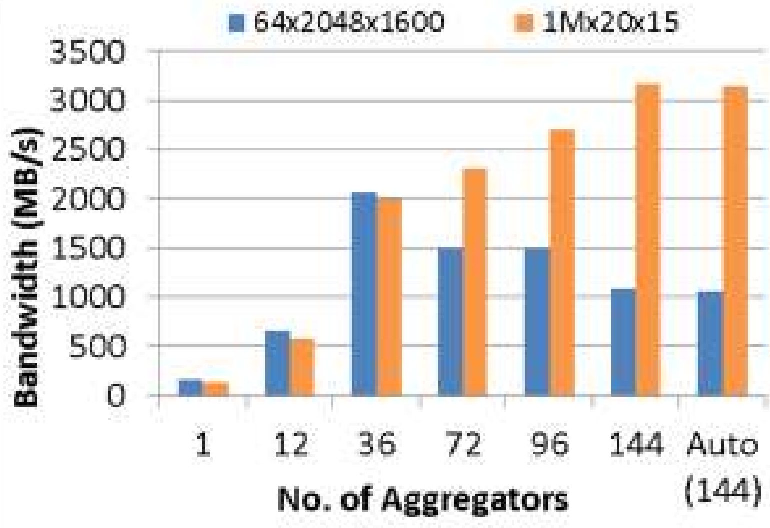
dynamic segmentation



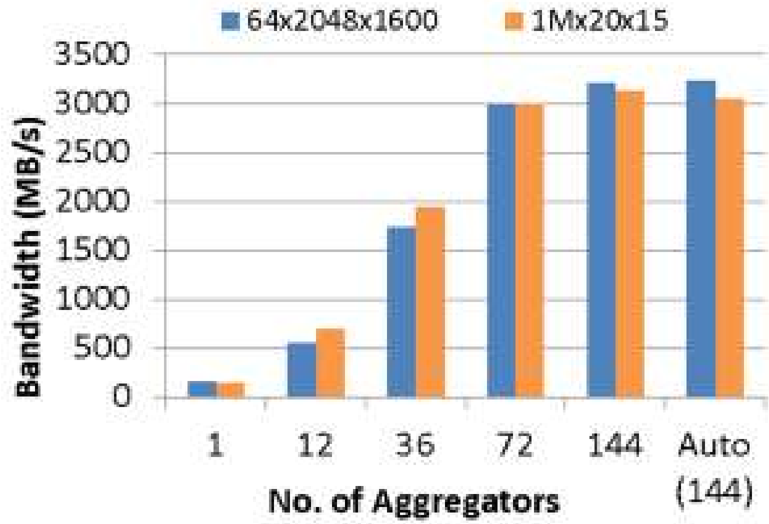
two-phase I/O

Deimos Tile I/O

- 144 processes test case



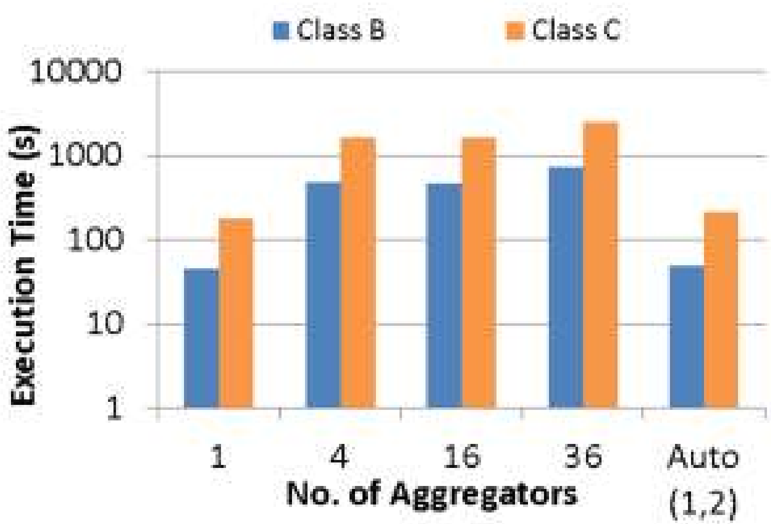
dynamic segmentation



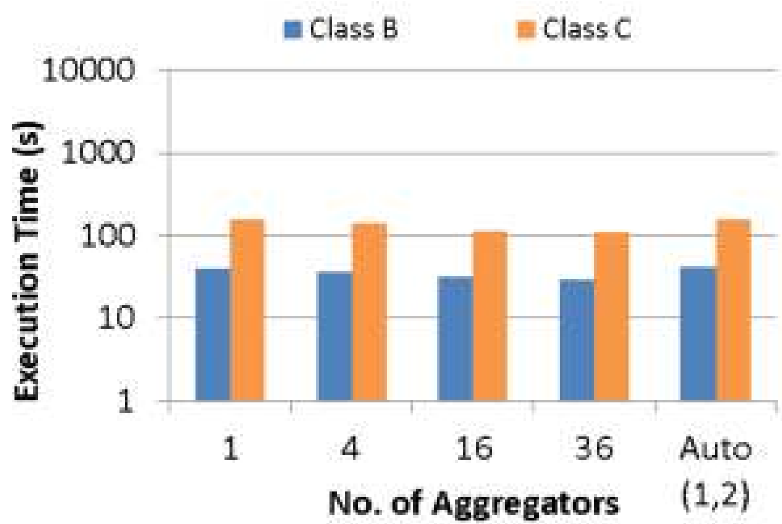
two-phase I/O

Deimos BT I/O

- 36 processes test case



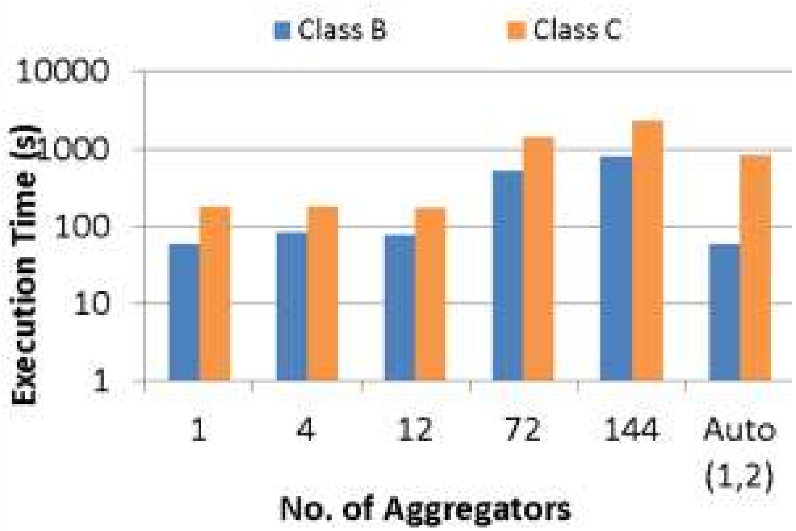
dynamic segmentation



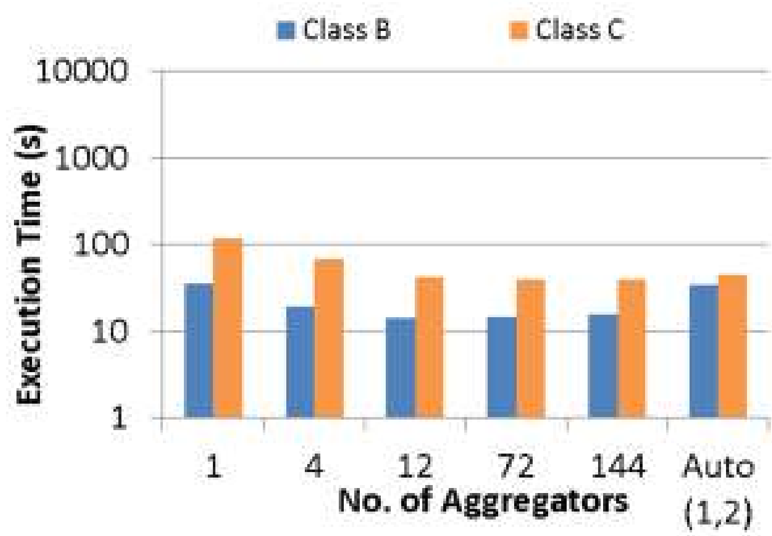
two-phase I/O

Deimos BT I/O

- 144 processes test case



dynamic segmentation



two-phase I/O

Discussion of results

- 134 tests executed in total
  - 88 tests lead to best or within 10% of optimal performance
  - 110 were within 25% of best performance

- Focusing on two-phase I/O algorithm only:
  - 29 out of 45 test cases outperformed one aggregator per node strategy on average by 41%

Conclusions

- Good performance for many test cases
  - Problems mostly by dynamic and static segmentation
  - Refining step can lead to strongly uneven size of groups
- Handling multiple cycles
  - *np * bytes per process >> na * k*

    -> *na = np*
- Would be good to know internally what is the factor restricting k
- Current implementation assumes uniform distribution of data across processes

Future work

- Fix known issues
- Extend work to read operations as well
- Re-work refining steps for dynamic and static segmentation algorithm

- Perform larger set of measurements
  - More real-world applications
  - More platforms, larger process counts etc.